ZQL: A Compiler for Privacy-Preserving Data Processing

Cédric Fournet
Microsoft Research

Markulf Kohlweiss
Microsoft Research

George Danezis
Microsoft Research

Zhengqin Luo
MSR-INRIA Joint Centre

Abstract

ZQL is a query language for expressing simple computations on private data. Its compiler produces code to certify data, perform client-side computations, and verify the correctness of their results. Under the hood, it synthesizes zero-knowledge protocols that guarantee both integrity of the query results and privacy for all other data.

We present the ZQL language, its compilation scheme down to concrete cryptography, and the security guarantees it provides. We report on a prototype compiler that produces F# and C++. We evaluate its performance on queries for smart-meter billing, for pay-as-you-drive insurance policies, and for location-based services.

1 Introduction

A variety of private user data is used to tailor modern services, and some go as far as billing based on fine grained customer readings. For example, smart meters are used to charge a different tariff depending on the time of electricity usage; pay-as-you-drive insurance premiums depend on detailed driving pattern of drivers. Such schemes are currently implemented by collecting fine-grained information, and processing it on the service side—an architecture that has led to serious privacy concerns.

This paper supports an alternative approach: clients could perform sensitive computations on their own data certified by meters or car on-board units [61, 66], and upload only the results, together with a proof of correctness to ensure their integrity. We propose ZQL, a simple query language to express at a high level such computations, without any cryptographic details. Queries are compiled to code for the data sources, the clients, and the verifiers by synthesizing zero-knowledge protocols.

The most popular language for querying and performing computations on user data is SQL [33] based on relational algebra. The ZQL feature set was chosen to support a subset of SQL. Data is organized into tables of rows, with private or public columns. Queries accept tables as inputs, and can iterate over them to produce other tables, or aggregate values. Simple arithmetic operations on rows are supported natively, and so is a limited form of SQL joins through lookups.

ZQL offers advantages over hand-crafted protocols, in that computations are flexible and can be expressed at a high level by application programmers. The computations can also be modified and recompiled, without the need to involve cryptography experts.

The ZQL compiler is free to synthesize custom zero-knowledge protocols behind the scene, and we currently support two main branches, for RSA and pairing based primitives. We also support a symbolic execution backend to derive estimates of the cost of performing and verifying queries. Synthesized protocols themselves are internally represented and optimized as fragments of an extended ZQL language until the final code is emitted. Intermediate ZQL is strongly typed, and precise refinement types can be used to verify security properties on the final compiled code, using F7 [18] or F* [64].

Informally, for a given source query, the desired security properties on the resulting ZQL-compiled code are:

- *Correctness.* For any given source inputs, the sequential composition of the cryptographic queries for the data sources, the user, and the verifier yields the same result as the source query.
- *Integrity.* An adversary given the capabilities of the user cannot get the verifier to accept any other result—except with a negligible probability.
- *Privacy.* An adversary given the capabilities of the verifier, able to choose any two collections of inputs such that the source query yields the same result, and given the result of the user's cryptographic query, cannot tell which of the two inputs was used.

This corresponds to the source query being executed by a fictional trusted third party sitting between the user and the verifier.

Contents The rest of the paper is organized as follows. §2 introduces our query language using a series of privacy-preserving data processing examples. §3 specifies our target privacy and integrity goals. §4 reviews the main cryptographic mechanisms used by our compiler. §5 describes the compilation process. §6 gives our main security theorems. §7 discusses applications and §8 provides experimental results and discusses future work.

Related work The ZQL language provides private data processing. The zero-knowledge protocols synthesized are standard Σ-protocols [36, 34, 37], but in ZQL they are used for proving the correctness of general computations rather than for cryptographic protocol design.

Arguably, previous works on zero-knowledge compilers focused on the latter as the primary use-case [23, 57, 4, 2]. The use of zero-knowledge for authentication and authorization as in credential and e-cash technologies [27, 57, 6] received particular attention, but, to our knowledge, no-one considered the use of Σ-protocols to prove the execution of general programs.

More specifically, a long line of work [23, 9, 3, 4] culminating in the *CACE* compiler tackles the problem of automatically translating proof goals specified in the Camenisch-Stadler notation [26] into efficient Σ-protocols. Intermediate translations steps of ZQL (the shared translation) are at a similar level of abstraction to the Camenisch-Stadler notation but ZQL also synthesizes those representations from code, and then proceeds to compile them to low level operations. *ZKPDL* [57] an alternative compiler for Σ-protocols, uses a natural language inspired specification of zero-knowledge proof goals. This specification language may be even closer in spirit to our intermediary notation, as it allows for the possibility to specify the generation of the protocol inputs. The authors of the CACE compiler discuss the difference and similarity between these two approaches in a Usenix poster [10]. The cryptographic prototyping language *Charm* [2] also includes a zero-knowledge proof compiler for Camenisch-Stadler statements which is currently primarily a proof of concept and thus less sophisticated than CACE and ZKPDL. We are also aware of an embedding of a zero-knowledge language in C++ [50].

ZQL differs from standard multi-party computation compilers [55], in that it assumes the client knows all private data. This assumption allows for single round protocols, and the efficient non-interactive implementation of non-linear operations including joins.

2 A Language for Private Data-processing

Why ZQL? We design ZQL to support privacy protocols that rely on client-side computation while requiring high integrity [39]. In this setting, a number of (possibly independent) sources of personal data are used as an input to a computation performed by a user. The results of the computation are sent to a service, while private input data is kept secret. The integrity is guaranteed by cryptographic proofs that establish the authenticity of the data sources and the correctness of the computation. The ZQL compiler is responsible for producing the code executed by the data sources, the prover and the verifier, as illustrated in Figure 1. Compiling allows us to verify the security of the resulting protocols using refinement types (see §6 and Appendix A) without the need to trust the compiler.

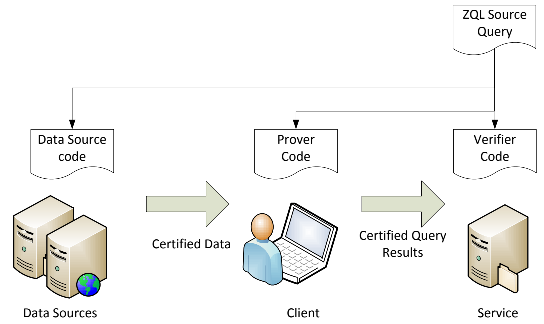

Figure 1: ZQL in a privacy friendly computation system.

There are advantages in de-coupling data sources from specific computations. It allows for meters, or services providing personal data, to remain simple, cheap, and generic. In turn, the computations, such as billing, can be updated without changing the devices that certify readings. Finally, private computations can aggregate disparate data sources that are not aware of one another, or may not trust one another with the privacy or integrity of the computations.

We first provide a brief description of our source language and then illustrate its primitives through simple examples. §7 provides larger examples of protocols that have been proposed in the literature.

The ZQL language At its core, ZQL is a pure expression language, with built-in operators that act on integers and tables. Figure 2 gives its abstract syntax. A query $\theta \rightarrow e$ consists of the declaration of input variables (θ) that can be either public or private, and of an expression body (e).

Expressions consist of variables, operators applied to sub-expressions $\tilde{e}$ (including constants as a special case when $\tilde{e}$ is empty), and let bindings for sequential composition. Expressions evaluate to tuples of values: for example, the expression **let** $x : int, y : int = e$ **in** e_0 first evaluates the sub-expression e to a pair of integers v_x, v_y, then evaluates e_0 after substituting v_x and v_y for x and y.

A variety of operators support arithmetic (0, 1, $+$, $*$), booleans ($=$, $\wedge$), and operations on tables (**map**, **fold**,

$$
\begin{array}{rcll}
e & ::= & & \text{Expressions} \\
 & | & x & \text{variable} \\
 & | & op\ \tilde{e} & \text{application} \\
 & | & \mathbf{let}\ \rho = e\ \mathbf{in}\ e & \text{let binding} \\
 & | & \downarrow e & \text{declassification} \\
op & ::= & & \text{Operators} \\
 & | & (\_,\_) \mid (\_,\_,\_) \mid \dots & \text{tuples} \\
 & | & + \mid - \mid * & \text{arithmetic} \\
 & | & = \mid \wedge & \text{boolean} \\
 & | & \mathbf{map}\ (\rho \rightarrow e) & \text{map iterator} \\
 & | & \mathbf{fold}\ (\rho \rightarrow e) & \text{fold iterator} \\
 & | & \mathbf{lookup}\ \rho & \text{table lookup} \\
 & & & \\
\tau & ::= & & \text{Types} \\
 & | & int\ [pub] & \text{security type} \\
 & | & \rho\ table & \text{table} \\
 & | & \rho\ lookuptable & \text{lookup table} \\
 & & & \\
\rho,\theta,\Gamma & ::= & \varepsilon \mid x : \tau\{\varphi\},\rho & \text{Tuple types} \\
 & & & \text{(binding } x \text{ in } \varphi \text{ and } \rho\text{)}
\end{array}
$$

Figure 2: ZQL Syntax

lookup). The iterators **map** and **fold** are parametrized by a ZQL expression, conceptually acting as the body of the corresponding loop. (We write these expressions as functions, but they can only specialize the iterator; they cannot be assigned to variables.)

Query inputs and expression results are specified using tuples of typed variables (θ for query inputs and ρ for sub-expressions). Each base type can be marked as public, and is otherwise treated as private. Types also include tables, where ρ indicates the type of each row in the table. Tables can contain mixtures of public and private columns; for example, (*time*:int pub, *reading*: int) table is the type of tables of private readings indexed by public times. On the other hand, the current ZQL compiler does not attempt to hide the query definition itself, or the number of rows in tables.

Intermediate expressions are automatically classified as public or private, depending on the types of their variables, following a standard information flow discipline: public inputs can flow to private results, but not the converse. Alternatively, a ZQL expression can be *explicitly declassified*, using the special operator $\downarrow e$ which specifies that the result of e can be released to the verifier, and marks it as public.

A ZQL query itself defines the privacy goals of the synthesized zero-knowledge protocols. For example, a query $\theta \rightarrow \downarrow e$ where e does not contain any declassification states that only the final result of the query is released, and that the protocol should not leak any side information on inputs marked as private in θ. A key feature of the language is that the underlying cryptographic mechanisms are totally hidden in the definition of the ZQL query. Since the ZQL query defines what results are declassified, it is important that users, or their proxies, review it to ensure no more than the necessary information leaks from it. Additional privacy mechanisms, such as differential privacy [41], could be used to measure or minimize any leakage resulting from the query declassification.

The ZQL language is strongly typed, with a type system simple enough to allow for automated type checking and type inference, which means that the programmer only has to write the input types of the query. We write $\Gamma \vdash e : \rho$ to state that expression e has type ρ in environment Γ. The type system ensures both *runtime safety*: e returns only to values of types ρ, and *non-interference*: in the absence of declassification, e does not leak inputs typed as private in Γ to results typed as public in ρ. We omit the formal definition of the language semantics and type system, which are standard. Internally, ZQL relies on a richer type system based on refinements types [16, 47] to keep track of various properties and to structure our security proofs—see §6.

ZQL by example We present the ZQL language and semantics through simple concrete examples, building to fuller queries that address problems in the literature in §7. The first example query computes the discriminant of the polynomial $xk^2 + yk + z$, for public x and private y and z.

let *discriminant* (x:int pub) (y:int) (z:int) = $\downarrow (z{*}z\ -\ 4{*}x{*}y)$

Anticipating on its compilation, the part of the expression that is linear in the secrets, namely $-4 * x * y$, can be proved efficiently through homomorphisms of Pedersen commitments, while the non-linear $z * z$ requires a Σ-protocol to prove the correctness of the private multiplication. The ZQL compiler will chose to synthesize the right proof techniques for each case, and linear expressions are compiled into efficient proofs.

The query declassifies its result, which leaks some information about y and z. For instance, given $x = 30$ and *discriminant x y z* = 1000, if the verifier knows a priori that $0 \leq y < 200$ and $0 \leq z < 200$, then it can infer that (y,z) is one of the pairs $(5,40)$, $(45,80)$, $(75,100)$ or $(155,140)$, but our privacy theorem ensures that its does not learn which pair was actually used.

Our next examples illustrate the use of tables and iterators **map** and **fold**. The first query computes the sum of all integers in table X, while the second returns the sum of their squares. The third query takes a table with a public column and two secret columns and returns a table with the same public column, and the element-wise sum of the secret columns. By design, the size of the tables is not hidden by ZQL.

let *sum_of_x* (X : int table) =
  $\downarrow$ (**fold** ((s, x) $\rightarrow s + x$) 0 X)
let *sum_of_square* (X : int table) =

$\downarrow$ (**fold** ((s, x) $\rightarrow s$ + $x{*}x$) 0 X)
let *linear* (T: (int pub $*$ int $*$ int) table) =
$\downarrow$ (**map** ((a,x,y) $\rightarrow a$, x+y) T)

The iterators are parametrized by a sub-query, which is applied to every row of the table, accumulating the sums in s, or building another table of results. The equivalent SQL statements would be select SUM(x) from X, select SUM(x*x) from X, and select a, x+y from T. The first and third queries compute linear combinations of secrets; we compile them without the use of any expensive Σ-protocols. We found sum queries to be frequent enough to justify some derived syntax: we write **sum** $\rho \rightarrow e\ T$ as syntactic sugar for **fold** $(s,\rho \rightarrow s+e)\ 0\ T$.

A key feature of the ZQL language is the ability to perform lookups on input tables. This provides a limited form of *join* and enable the computation of arbitrary functions with small domains. The expression **lookup** $x\ T$ finds a row $x, v_1, \dots v_n$ in T that matches x, and returns $v_1, \dots, v_n$. From an information-flow viewpoint, the result of a lookup on a private variable is private (even if the lookup table is public); in that case, ZQL leaks no information about which row is returned. If multiple rows match, the verifier is only able to assert that any matching row was used. If no row matches a runtime exception is raised on the prover side, and the proof fails. This semantics allow the implementation of functions, set membership tests, and half-joins. To enable lookups, input tables currently need to be signed using a re-randomizable signature by a trusted source, so lookups on intermediate, computed tables are not supported.

The example *blur*, listed below, repeatedly uses a lookup to map private city identifiers to their respective countries; the resulting table is then declassified.

let *blur* (X: int table) (F: (int $*$ int) lookuptable) =
$\downarrow$ (**map** (*city* $\rightarrow$ **lookup** *city* F) X)

The equivalent SQL statement would be select F.country from X, F where F.city = X.city. The query implementation relies on a data source that issues a signed table from cities to countries.

3 Security

The next two sections provide rigorous security definitions for what the ZQL compiler achieves and the cryptographic building blocks it uses, necessary for formulating our security theorems in §6. The mere fact that we can give formal cryptographic definitions for what is in fact an infinite set of cryptographic protocols relies on our simple expression language having a formal semantic for both source and compiled programs. Readers interested in compiler architecture can jump straight to §5, or those curious about applications can find them in §7.

Notations Consider a well-typed ZQL source query $\mathsf{Q} \overset{\triangle}{=} \theta \rightarrow \downarrow e$, with t input variables $\theta = (x_i : \tau_i)_{i=0..t-1}$, that declassifies only its result. As explained in §2, the typed variables θ specify the data sources and privacy policy. Let $\vec{T}$ range over values of types θ, and $R = \mathsf{Q}(\vec{T})$ be the corresponding query result. Given Q, our compiler produces queries $(\mathsf{S}, (\mathsf{K}_i, \mathsf{D}_i)_{i=0..t-1}, \mathsf{P}, \mathsf{V})$ with formal parameters indicated in parentheses as follows. (We use primed variables for compiled values.)

- S, the setup generator, generates global parameters χ used by the commitment scheme;
- $(\mathsf{K}_i)_{i=0..t-1}$, the data sources key generation, generate key pairs $sk_i, vk_i := \mathsf{K}_i(\chi)$ used to sign data;
- $(\mathsf{D}_i)_{i=0..t-1}$, the data sources, extend and sign each input: $T_i' := \mathsf{D}_i(\chi, sk_i, T_i)$;
- P, the prover, produces an extended result from extended inputs: $R' := \mathsf{P}(\chi, \vec{vk}, \vec{T}')$;
- V, the verifier, returns either some source result $R = \mathsf{V}(\chi, \vec{vk}, R')$ or a verification error.

Main Properties We first define functional correctness when all participants comply with the protocol.

Definition 1 $(\mathsf{S}, (\mathsf{K}_i, \mathsf{D}_i)_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$ *correctly implements the source query* Q *when, for any source inputs* $\vec{T} : \theta$ *and* $\chi := \mathsf{S}$, $(sk_i, vk_i := \mathsf{K}_i(\chi))_{i=0..\ell-1}$, *we have*

$$\mathsf{V}(\chi, \vec{vk}, \mathsf{P}(\chi, \vec{vk}, \mathsf{D}_i(\chi, sk_i, T_i)_{i=0..\ell-1})) = \mathsf{Q}(\vec{T}).$$

We define privacy as indistinguishability between two series of chosen inputs that yield the same query result.

Definition 2 *Given a source query* Q *and an adversary* $\mathscr{A}$, *let* $\mathsf{Adv}^{Priv}_{\mathscr{A}} = |2\Pr[\mathscr{A}\ wins] - 1|$ *where the event* '$\mathscr{A}$ wins' *is defined by the following game:*

1. *The challenger runs* S *and* K_i *to generate setup* χ *and keys* $\vec{sk}$, $\vec{vk}$*; it provides* χ *and* $\vec{vk}$ *to* $\mathscr{A}$.
2. *The adversary* $\mathscr{A}$ *provides two vectors of input data* $\vec{T}^0 : \theta$ *and* $\vec{T}^1 : \theta$ *such that (a) they coincide on public data and (b)* $\mathsf{Q}(\vec{T}^0) = \mathsf{Q}(\vec{T}^1)$.
3. *The challenger picks a random bit b, encodes the inputs* $T_i' := D_i(\chi, sk_i, T_i^b)$ *for* $i = 0..\ell-1$, *and generates* $R' := P(\chi, \vec{vk}, T_i')$.
4. *Given* R', $\mathscr{A}$ *returns his guess* b', *and* wins *iff* $b = b'$.

$(\mathsf{S}, (\mathsf{K}_i)_{i=0..\ell-1}, (\mathsf{D})_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$ *is* (t, ε)*-private when, for all* $\mathscr{A}$ *running at most for time t, we have* $\mathsf{Adv}^{Priv}_{\mathscr{A}} \leq \varepsilon$.

Note that we do *not* formally provide privacy protection against corrupted data sources.[1]

We define integrity as a game in which an adversary has to produce an invalid but accepted response.

[1] To strengthen our scheme against data source attacks, we would have to rerandomize all cryptographic material flowing from data sources to verifiers, which precludes our efficient use of homomorphic commitments.

Definition 3 *Given a source query* Q *and an adversary* $\mathscr{A}$*, let* $\mathsf{Adv}^{Snd}_{\mathscr{A}} = \Pr[\mathscr{A}$ *wins*$]$ *where the event '*$\mathscr{A}$ wins*' is defined by the following game:*

(1) *The challenger runs* S *and* K_i *to generate setup* χ *and keys* $\vec{sk}$*,* $\vec{vk}$*; it provides* χ *and* $\vec{vk}$ *to* $\mathscr{A}$*.*

(2) *The adversary* $\mathscr{A}$ *can adaptively corrupt a data source* D_i *to get its signing key* sk_i *and, at the same time, can obtain signed inputs* $T_i' := D_i(\chi, sk_i, T_i)$ *for source inputs* $T_i : \tau_i$ *of its choice.*

(3) *Valid results are values* $R = Q(\vec{T})$ *such that, for each i, either i was corrupted or* T_i *was signed. The adversary wins if he outputs* R' *such that* $V(\chi, \vec{vk}, R')$ *returns any invalid result* $R^\star$*.*

$(\mathsf{S}, (\mathsf{K}_i)_{i=0..\ell-1}, (\mathsf{D})_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$ *is* (t, ε)*-sound when, for all* $\mathscr{A}$ *running at most for time t, we have* $\mathsf{Adv}^{Snd}_{\mathscr{A}} \leq \varepsilon$*.*

Depending on the adversary, there can be zero, one, or numerous valid responses. In fact, depending on the query and the input tables, whether a response is valid may not even be efficiently checkable. The definition is, however, still meaningful.

4 Main Cryptographic Tools (Review)

Signatures A *digital signature scheme* allows everyone in possession of the verification key *vk* to verify the authenticity of data signed by the owner of the corresponding signing key *sk*. We use signatures to let verifiers authenticate our data sources. Instead of signing private data in the clear, data sources sign public commitments ; thus, the resulting signature tags are also public.

Cryptographic groups Besides conventional digital signatures, for which we use standardized schemes, our remaining cryptographic tools can either be specified for composite order groups obtained by computing module an RSA modulus or for prime order groups with a bilinear pairing. We use the latter for our presentation and formal analysis as it offers both performance and conceptual advantages.

Let G, $\hat{G}$, and G_T be groups of prime order q. Let $g \in G$ and $\hat{g} \in \hat{G}$ be generators of G and $\hat{G}$ respectively. A bilinear pairing is an efficiently computable function $\hat{e} : G * \hat{G} \rightarrow G_T$ that is bilinear, i.e. $\forall a, b \in \mathbb{F}_q : e(g^a, \hat{g}^b) = e(g, \hat{g})^{ab}$ and non-degenerate, i.e. $e(g, \hat{g}) \neq 1$. Whenever possible we perform all operations in the base group G with the shortest representation.

Commitments A *commitment scheme* allows a user to *commit* to a hidden value such that he can *reveal* the committed value at a later stage. The properties of a commitment scheme are *hiding*: the committed value must remain hidden until the reveal stage, and *binding*: the only value which may be revealed is the one that was chosen in the commit stage. We use the perfectly hiding commitment scheme proposed by Pedersen [59]: Given a group G of prime order q with generators g and h, generate a commitment C_x to $x \in \mathbb{F}_q$ by sampling a random opening $ox \leftarrow \mathbb{F}_q$ and computing $C_x = g^x h^{ox}$. The commitment is opened by revealing x and ox.

Two useful properties of Pedersen commitments are (i) their *homomorphic property* that allows to derive a commitment to the linear combination of input values and (ii) their *algebraic structure* that allows for efficient zero-knowledge proof. For RSA groups, we use commitments with similar properties [48, 38].

Zero-knowledge proofs [65, 44, 13] provide a verifying algorithm with an efficient means for checking the truth of a statement by guaranteeing that given access to a successful proof generation algorithm one can extract a secret witness for said truth. At the same time, *zero-knowledge proofs* [52, 51], and the related concepts of witness indistinguishable proofs [43, 36], allow the prover to keep this witness secret. We make use of a long line of work on efficient proofs of conjunctions of discrete logarithm (DL) representations [63, 32, 58, 36, 34, 21, 30, 37, 56]. For non-linear computations such as multiplication, we use the approach of Brands [21], Camenisch [30], and Cramer and Damgård [35].

DL representation proofs are interactive protocols of three or more rounds. To ease deployment and minimize communications, we use the Fiat-Shamir Heuristic [45] and replace messages sent by the verifier with hash function computations. The resulting protocols can still be formally analyzed in the random oracle model [14, 68].

Proof compatible signatures The combination of zero-knowledge proofs and digital signatures allows us to prove authentication properties on private data, such as, for instance, the existence and properties of a matching row when performing a private lookup.

We use CL-signatures [24], as they are compatible with DL representation proofs. The original scheme was proven secure under the Strong RSA assumption and requires groups with hidden order [8, 28]. Other CL-signature proposals rely on a variety of assumptions based on bilinear pairings [25, 20, 5, 29] and require more standard prime order DL-representation proofs.

To certify our lookup tables, data sources extend each row of the table with a CL-signature. For instance, tables of triples of integers x_0, x_1, x_2 are extended to tables with rows of the form (x_0, x_1, x_2, e, v, A). The verification equations for RSA and bilinear pairing based CL-signatures are of the form $Z = A^e R_0^{x_0} R_1^{x_1} R_2^{x_2} S^v$ and $\hat{e}(Z, \hat{g}) = \hat{e}(A, pk * g^e)\hat{e}(R_0^{x_0} R_1^{x_1} R_2^{x_2} S^v, \hat{g})$ respectively, where $(Z, R_0, R_1, R_2, S, pk)$ are group elements that form the components of the verification key vk. Both verification equations can be proven using efficient DL representations. The security of these two schemes is based on the *strong RSA* assumption and the *strong Diffie-Hellman* (SDH) assumption respectively.

5 Compiler Architecture

Protocol Overview The ZQL compiler takes a source query, which contains no cryptographic computations, and automatically produces programs for the the data sources, the prover, and the verifier.

First, the compiler augments the source query with cryptographic commitments to secrets and representation equations to generate a *shared translation* that will lead to both prover and verifier code. Some commitments are computed and signed by the data sources that certify the computation inputs, and simply passed to the prover and verifier programs. Others, representing intermediate secrets in the query, are interleaved with the source computation: for any such secret x, the prover may sample a secret opening ox, compute a Pedersen commitment $C_x =_G g^x h^{ox}$, and send it to the prover; and the prover may check it using a zero-knowledge proof.

Linear relations between secrets do not require complex zero knowledge proofs, as they can be checked by the verifier simply by using the homomorphisms of Pedersen commitments. For example, a private sum $z = x + y$ will have commitment $C_z =_G C_x * C_y$. Such commitments need not be transmitted, as they can be recomputed by the verifier. On the other hand, non-linear relations between secrets, including multiplication and table lookup, require Σ-protocol proofs to be synthesized. For instance, to prove that z is the product of a secret x committed in C_x and a secret y, one proves the conjunction of the representation equations $C_x =_G g^x h^{ox}$ and $1 =_G (C_x)^{-y} g^z h^{oz}$. Note that the second equation uses a variable commitment C_x as a base.

All Σ-protocols used in the compiler come down to proving knowledge of representations of the secret values underlying the secret discrete logarithm representations of public group elements, and equality relations between the secret values. Assume the ZQL query reduces to proving in zero-knowledge the representations $\vec{C} =_G \vec{e}[\tilde{x}]$ of a number of commitments $\vec{C}$, represented by public group elements, using a number of secrets $\tilde{x}$ (including secret openings). For the multiplication example above, we have two equations on five secrets: $\vec{C} \equiv (C_x, 1)$, $\tilde{x} \equiv (x, ox, y, z, oz)$ and $\vec{e}[(\alpha, \beta, \gamma, \delta, \varepsilon)] \equiv (g^\alpha * h^\beta, C_x^{-\gamma} g^\delta h^\varepsilon)$. The zero-knowledge protocol synthesized works as follows. The prover

(1) samples a vector of random values $\tilde{t}$, one for each secret in $\tilde{x}$; We call $\tilde{t}$ values the proof randomness;

(2) computes the challenge $c = H(\vec{e}[\tilde{t}])$;

(3) computes the responses $\tilde{r} = \tilde{t} - c * \tilde{x}$, for all secrets.

The proof sent to the verifier consists of the public parameters and values, the commitments $\vec{C}$, the global challenge c, and the responses $\tilde{r}$. The verifier checks that $H(\vec{C})^c *_G \vec{e}[\tilde{r}]))) = c$. This ensures that the prover knows the secret values in the commitments [45, 14]. As detailed below, our compiled prover and verifier programs introduce secrets and process equations on the fly, depending on the query and its inputs.

Once the shared translation is decided, the specialization into a prover and verifier is relatively straightforward. It involves mainly ensuring the right data flows within the query processing to compute all commitments, challenges, and responses, and to correctly verify them in the same order. The inputs of the shared translation also determine the data source programs that generate keys, compute commitments, and sign extended data.

Embedding cryptography within ZQL Our compiler mostly operates within ZQL, with F# and C++ back-ends to turn the compiled queries into executable code. This enables us to reason about code in a simple, domain-specific language. To this end, Figure 3 supplements the source language of Figure 2 with the types and operators for expressing cryptographic operations. Expressions are extended with **assert**, used in the shared translation to embed proof obligations. As an invariant, all asserted equations φ must hold at runtime. We have types and operations for integers modulo q ($\mathbb{F}_q$, written *num*), for group elements (elt_G), and for bitstrings, and more specific sub-types to keep track of their usage. For instance, *hash* is the sub-type of bitstrings representing cryptographic hashes, and *x opening* is a sub-type of *num* tracking openings generated for x. In our presentation, we use standard abbreviated forms for their operations; for instance we often omit group parameters, writing g^x for $exp_G\ g\ x$.

The abstract setup S produces global parameters χ supplied by our cryptographic runtimes, including q, the prime order of G, $\hat{G}$, and G_T; and independent, random generators g, h, $(R_i)_{i=0..n}$, S, Z in G; and $\hat{g}$ in $\hat{G}$. Its fixed code is provided by our cryptographic libraries.

We use $D_{LT} \subseteq 0..\ell - 1$ to denote the subset of data sources that sign lookup table. The key generation K_i is defined as *keygen* χ when τ_i is a scalar or a table ($i \notin D_{LT}$), and as the CL-key generation **let** $sk = sample()$ **in** $sk, (\hat{g})^{sk}$ when τ_i is a lookup table ($i \in D_{LT}$). The data source code D_i is explained below, as we discuss these two representations.

Shared Translation We extend the source query with openings and commitments, but not yet with the corresponding proof randomness and responses.

The main difficulty of the translation is to select cryptographic mechanisms, and notably intermediate com-

e	$::=$	$\dots$	Expressions
	$\mid$	$\textbf{assert}\ \varphi; e$	static assertion
op	$::=$	$\dots$	Operators
	$\mid$	$-1, 0, 1, \dots$	constants
	$\mid$	$\mathit{sample} \mid \mathit{random} \mid \mathit{div}$	exponents (mod q)
	$\mid$	$*_G \mid =_G \mid \mathit{exp}_G$	group operations
	$\mid$	$\hat{e} : G * \hat{G} \to G_T$	EC bilinear form
	$\mid$	$\mathit{extend} \mid \mathit{finalize}$	cryptographic hash
	$\mid$	$\mathit{keygen} \mid \mathit{sign} \mid \mathit{verify}$	plain signatures
	$\mid$	$\textbf{mapP}\ \_ \mid \textbf{mapV}\ \_$	
	$\mid$	$\textbf{foldP}\ \_ \mid \textbf{foldV}\ \_$	translated iterators
τ	$::=$	$\dots$	Types
	$\mid$	$\mathit{num} \mid x\,\mathit{opening} \mid x\,\mathit{rand}$	exponents (mod q)
	$\mid$	$\mathit{elt}_G \mid x\,ox\,\mathit{commitment}$	group elements
	$\mid$	hash	cryptographic hash
	$\mid$	$\mathit{tag}_i \mid \mathit{sk}_i \mid \mathit{vk}_i$	plain signatures

Figure 3: ZQL internal constructs

$$\begin{aligned}
\llbracket x : \tau\{\varphi\} \rrbracket &= x : \tau\{\varphi\} \text{ when } \tau \text{ is public;} && (1)\\
&\quad \text{otherwise:}\\
\llbracket x : \mathit{int}\{\varphi\} \rrbracket &= x : \mathit{int}\{\varphi\}, && (2)\\
&\quad ox : x\ \mathit{opening},\\
&\quad C_x : x\,o\ \mathit{commitment}\\
\llbracket \rho\ \mathit{table} \rrbracket &= \llbracket \rho \rrbracket\ \mathit{table}, s : \mathit{tag}\\
\llbracket \rho\ \mathit{lookuptable} \rrbracket &= (\rho, \sigma)\ \mathit{table}\\
\sigma &= e : \mathit{num}, v : \mathit{num}, A : \mathit{elt}_G\\
\llbracket \varepsilon \rrbracket &= \varepsilon\\
\llbracket x : \tau\{\varphi\}, \rho \rrbracket &= \llbracket x : \tau\{\varphi\} \rrbracket, \llbracket \rho \rrbracket
\end{aligned}$$

Figure 4: Shared translation of types and environments

mitments, to run the private computation: for every private sub-expression, our compiled protocol may rely on zero, one, or more Pedersen openings and commitments, and it may allocate some proof randomness or not.

In this presentation, for simplicity, we give a formal translation that assumes that *all* source private integer variables are handled uniformally, with a commitment in the same group sharing the same bases, and (later) with a proof randomness for the secret and for its opening. Figure 4 and 5 show how we translate types and expressions, respectively, in this special case. We discuss our general, more efficient compilation scheme below.

A source expression is *public* in a typing environment when all its free variables have public types. The translation leaves public types (1) and expressions (3) unchanged. The translation of a private integer expression is a triple of an integer for the source value, its opening, and its commitment, with the types given on line (2).

$$\begin{aligned}
\llbracket \Gamma \vdash e \rrbracket &= e \text{ when } e \text{ is public;} && (3)\\
&\quad \text{otherwise:}\\
\llbracket \Gamma \vdash x \rrbracket &= \llbracket \Gamma(x) \rrbracket\\
\llbracket \Gamma \vdash \textbf{let}\ \rho = e\ \textbf{in}\ e_0 \rrbracket &= && (4)\\
&\quad \textbf{let}\ \llbracket \rho \rrbracket = \llbracket \Gamma \vdash e \rrbracket\ \textbf{in}\ \llbracket \Gamma, \rho \vdash e_0 \rrbracket\\
\llbracket \Gamma \vdash \textbf{map}\,(\rho \to e)\,T \rrbracket &= && (5)\\
&\quad \textbf{map}\,(\llbracket \rho \rrbracket \to \llbracket \Gamma, \rho \vdash e \rrbracket)\,\llbracket \Gamma \vdash T \rrbracket\\
&\quad \text{where } \Gamma(T) = \rho\,\mathit{table}\\
&\quad \text{and } \Gamma, \rho \vdash e : \rho'\\
\llbracket \Gamma \vdash \textbf{fold}\,(a : \tau, \rho \to e)\,a\,T \rrbracket &= && (6)\\
&\quad \textbf{fold}\,(\llbracket a : \tau, \rho \rrbracket \to \llbracket \Gamma, a : \tau, \rho \vdash e \rrbracket)\\
&\qquad\quad \llbracket \Gamma \vdash a \rrbracket\ \llbracket \Gamma \vdash T \rrbracket\\
&\quad \text{where } \Gamma(T) = \rho\,\mathit{table}\\
&\quad \text{and } \Gamma, a : \tau, \rho \vdash e : \tau\\
\llbracket \Gamma \vdash a_0 + \textstyle\sum_{i=1}^n a_i * x_i \rrbracket &= && (7)\\
&\quad a_0 + \textstyle\sum_{i=1}^n a_i * x_i,\\
&\quad \textstyle\sum_{i=1}^n a_i * ox_i,\\
&\quad g^{a_0} *_G \textstyle\prod_{G,i=1}^n (C_{x_i})^{a_i}\\
&\quad \text{when the } a_i \text{ are public and the } x_i \text{ private}\\
\llbracket \Gamma \vdash x * y \rrbracket &= && (8)\\
&\quad \textbf{let}\ p : \mathit{int} = x * y\ \textbf{in}\\
&\quad \textbf{let}\ o' : \mathit{num} = ox * y\ \textbf{in}\\
&\quad \textbf{assert}\ 1 = (C_x)^y *_G g^{-p} *_G h^{-o'};\\
&\quad \textbf{Commit}\ p\\
&\quad \text{when } x \text{ and } y \text{ private}\\
\llbracket \Gamma \vdash \mathit{lookup}\ x_0\ T_i \rrbracket &= && (9)\\
&\quad \textbf{let}\ x_1, \dots, x_n, e, v, A = \mathit{lookup}\ x_0\ T_i\ \textbf{in}\\
&\quad \textbf{let}\ d, od, C_d = \textbf{Commit}\ (\mathit{random}())\ \textbf{in}\\
&\quad \textbf{let}\ p = d * e\ \textbf{in}\\
&\quad \textbf{let}\ o' = od * e\ \textbf{in}\\
&\quad \textbf{assert}\ 1 =_G {C_d}^e g^{-p} h^{-o'}\\
&\quad \textbf{let}\ A' = A * h^{-d}\ \textbf{in}\\
&\quad \textbf{assert}\ \hat{e}(Z, \hat{g})\hat{e}(1/A', pk_i) =_{G_T}\\
&\qquad (\textstyle\prod_{i=0}^n \hat{e}(R_i, \hat{g})^{x_i})\hat{e}(A', \hat{g})^e\\
&\qquad \hat{e}(S, \hat{g})^v \hat{e}(h, \hat{g})^p \hat{e}(h, pk_i)^d\\
&\quad \textbf{Commit}\ x_1, \dots, \textbf{Commit}\ x_n\\
&\quad \text{where } \Gamma(T_i) = (x_i : \mathit{int})_{i \in 0..n-1}\ \mathit{lookuptable}.\\
\llbracket \Gamma \vdash \downarrow x \rrbracket &= \downarrow x \text{ when } x \text{ private} && (10)
\end{aligned}$$

Figure 5: Shared translation of typed source expressions

Fresh commitments Our compilation rules may require openings and commitments on their arguments, and may not produce openings and commitments on their results. Our compiler attempts to minimize those cases. Nonetheless, assuming for instance that we need a commitment, we produce it on demand, using the expression abbreviation **Commit** below

$$\begin{array}{l}
\textbf{Commit}\ z \triangleq\\
\quad \textbf{let}\ oz : z\ \mathit{opening} = \mathit{sample}()\\
\quad \textbf{let}\ C_z : z\,oz\ \mathit{commitment} = g^z *_G h^{oz}\\
\quad \textbf{assert}\ C_z = g^z *_G h^{oz};\\
\quad z, oz, C_z
\end{array}$$

The translation is compositional, as can be seen on lines (4,5,6) in the figure. For instance, we translate **let** expressions by translating their two sub-expressions, and we translate source maps to maps that operate on their translated arguments.

The translation assumes prior rewriting of the source query into simpler sub-expressions. For instance, to compile the discriminant query of §2, we first introduce intermediate variables for the private product and the declassification, rewriting expression $\downarrow (z*z-4*x*y)$ into

$$e_d \overset{\triangle}{=} \textbf{let } p = z*z \textbf{ in let } d = p-4*x*y \textbf{ in} \downarrow d.$$

As a sanity check, our translation preserves typing, in an environment extended with the constants used in our cryptographic libraries; variants of this lemma with more precise refinement types for the prover and verifier translation can be used to verify their privacy and integrity.

Lemma 1 (Typing the shared translation) Let $\Gamma_0 \overset{\triangle}{=} g,h,Z,R_0,\dots R_n,S : elt_G, \hat{g}, (pk_i)_{i\in D_{LT}} : elt_{\hat{G}}$. If $\Gamma \vdash e : \rho$, then $\Gamma_0, [\![\Gamma]\!] \vdash [\![\Gamma \vdash e]\!] : [\![\rho]\!]$.

Next, we explain and illustrate the base cases of the shared translation on private expressions.

Expressions affine in private variables are translated by supplementing the expression with a linear expression on openings and an homomorphic product of commitments (7); we easily check that the resulting triple (z, oz, C_z) is such that $C_z = g^z *_G h^{oz}$. Note that the public constant a_0 is not included in the opening computation.

Expressions polynomial in private variables are translated using an auxiliary representation equation for every product of private expressions, depending on the availability of openings and commitments—see translation rule (8). To illustrate affine and quadratic expressions, let us translate the discriminant query $\theta \to \downarrow (e_d)$ where the source environment $\theta = x : int\, pub, y : int, z : int$ specifies that x is public, whereas y and z are private. By definition, the translated environment $[\![\theta]\!]$ is

$$\begin{aligned} &x : int\, pub,\\ &y : int, oy : \ y\, opening, C_y : y\, oy\, commitment,\\ &z : int, oz : \ z\, opening, C_z : z\, oz\, commitment \end{aligned}$$

and, as a translation invariant, we already know that $C_y =_G g^y h^{oy}$ and $C_z =_G g^z h^{oz}$. Applying rules (4), (8), (7), and (10) and inlining the definition of **Commit** we arrive at the shared translation

```
let $p$, $op$ , $C_p$ =
    let $p = z*z$
    let $o' = oz*z$
    assert $1 = (C_z)^z *_G g^{-p} *_G h^{-o'}$;                ($E_1$)
    let $op = sample()$
    let $C_p = g^p *_G h^{op}$
    assert $C_p = g^p *_G h^{op}$;                              ($E_2$)
    ($p$, $op$, $C_p$)
let $d$, $od$ , $C_d$ =
    $(p-4*x*y), (op-4*x*oy), (C_p * C_y^{-4*x})$
$\downarrow d$
```

and we easily check that C_d is a commitment to $z^2 - 4xy$ with opening $op - 4x*oy$. The code of the shared translation makes explicit the two representation equations for the private multiplication, presented more abstractly at the beginning of the section. Anticipating on the next stages of the translation, the prover will *compute* C_p, pass it to the verifier, and extend its challenge computation with equation E_2, whereas the verifier will receive some C_p and use it to check this equation. Note that the cryptographic overhead depends on the target level of privacy: given instead a source environment θ declaring that x is also private, the same discriminant expression would involve representation proofs for two private products.

Private lookups are translated using proofs of knowledge of signatures. To enable this, data sources extend input tables $T : \rho\ lookuptable$, where ρ is of the form $x_0 : int, \dots, x_n : int$, into tables $T' : (\rho, \sigma)table$ with a CL-signature at the end of each row, as follows:

```
$\mathsf{D}_i \overset{\triangle}{=} \chi, sk, T \to$ map ($x_0 \dots x_n \to$
                    let $e = random()$
                    let $v = random()$
                    let $A = (\prod_{G,i=0}^{n} R_i^{x_i} S^v Z^{-1})^{\frac{1}{sk+e}}$
                    $x_0, \dots, x_n, e, v, A$)
                 $T$
```

Although this pre-processing may be expensive for large tables, it can be amortized over many queries.

A lookup within a source query, such as the one from the *blur* query of §2, is translated to a proof of possession of a CL signature. For instance, let us translate the expression **lookup** $c\ F$ in environment $\rho = F : (city : int, country : int)\ lookuptable, city : int$. The environment is first translated to

$$\begin{aligned} [\![\rho]\!] = \quad &(city : int, country : int, \sigma)table,\\ &c : int, oc : c\ opening, C_c : c\ oc\ commitment \end{aligned}$$

The lookup itself is translated (using rule 9) to

```
$[\![\Gamma \vdash$ lookup $c\ F]\!]$=
    let $country$, $e$,$v$,$A$ = lookup $c\ F$
    let $d$, $od$, $C_d$ = Commit($random()$)
    let $p$, $o'$ = $d*e$, $od*e$
    assert $1 =_G {C_d}^e g^{-p} h^{-o'}$
    let $A' = A * h^{-d}$
    assert $\hat{e}(Z,\hat{g}) \cdot \hat{e}(1/A', pk_i) =_{G_T}$
            $\hat{e}(R_0,\hat{g})^c \cdot \hat{e}(R_1,\hat{g})^{country} \cdot$
            $\hat{e}(A',\hat{g})^e \cdot \hat{e}(S,\hat{g})^v \cdot \hat{e}(h,\hat{g})^p \cdot \hat{e}(h,pk_i)^d$
    Commit($country$)
```

This code first looks for a signed tuple $(city, country, e, v, A)$ in F such that $c = city$ and retrieves the remaining elements; it then proves knowledge of this tuple, without revealing which tuple is used in the proof, by blinding the element A of the signature. (Note that this proof internally relies on a proof of multiplication.)

Iterators and Committed Tables ZQL supports tables with mixed public and private columns, as well as iterators **map** and **fold**. To enable processing on their private contents, data sources extend tables with commitments and sign them. For instance, here is the code for the provider of the table of cities for the *blur* query.

```
$\mathsf{D}_i \stackrel{\triangle}{=} \chi, sk, X \rightarrow$
    $\textbf{let}\ X' = \textbf{map}\ (x\!: \mathrm{int} \rightarrow \textbf{Commit}\ x)\ X$
    $\textbf{let}\ H = \textbf{fold}\ (H, x, ox, C_x \rightarrow extend\ H\ C_x)\ H_0\ X'$
    $X', sign\ sk\ H$
```

This code first uses **map** to extend each source integer with a fresh opening and commitment, using the **Commit** abbreviation ; this yield the extended table X' passed to the prover; it then uses **fold** to computes the joint hash of these commitments and signs the result. (In the hash computation, H_0 is some fixed tag, and we omit a conversion from elt_G to *hash*). As outlined at the end of this section, both the prover and the verifier perform some initial processing for these extended tables: the prover must prove show his knowledge of the representation for these commitments, and the verifier must verify the signature and the representation proofs for these commitments.

We illustrate the translation of iterator **map** (5) on the *blur* query from §2. The translation of **fold** (6) is similar. The map expression of *blur* is translated to another map expression, in a translated environment that provides the extended input $X : \llbracket x : int \rrbracket\ table$:

$$\begin{aligned}&\llbracket \Gamma \vdash \textbf{map}(c \rightarrow lookup\ c\ F)\ X \rrbracket = \\ &\quad \textbf{map}\,(c, oc, C_c \rightarrow \llbracket \Gamma, c : int \vdash lookup\ c\ F \rrbracket)\, X\end{aligned}$$

and the translation continues with the **lookup** expression, as explained above.

Prover Translation Continuing from the result of the shared translation, the prover translation uniformly turns its assertions into a custom non-interactive Σ-protocol, in two passes, written $\llbracket \_ \rrbracket_1$ and $\llbracket \_ \rrbracket_2$, that produce code for the message randomness and for the responses, respectively.

Figure 6 defines these two passes, as well as the top-level query translation $\llbracket \_ \rrbracket_{\mathsf{PROVER}}$ that combines $\llbracket \_ \rrbracket_1$ and $\llbracket \_ \rrbracket_2$ with additional glue. Overall, the prover for a source query $\theta \rightarrow e$ is thus defined using this translation after the shared translation: $\mathsf{P} \stackrel{\triangle}{=} \llbracket\, \llbracket \theta \vdash e \rrbracket\, \rrbracket_{\mathsf{PROVER}}$.

First-message translation In the first pass, H is the public hash incrementally computing the global challenge, a is the accumulated cryptographic evidence that will be sent to the verifier, and every private variable x is replaced with a pair x, t_x where t_x is the proof randomness. (Openings ox are treated as any other secrets.) In combination with the shared translation, every private source expression becomes a tuple of the form $\llbracket \llbracket e \rrbracket \rrbracket_1 : (x, t_x, ox, t_{ox}, C_x)$ where x is the source value, t_x is the proof randomness for x, ox is an opening for x, t_{ox} the proof randomness for ox, and C_x is a commitment to x. For efficiency, all these additional values are optional in our compiler.

Compositionally, the type translation $\llbracket \rho \rrbracket_1$ maps shared environments to environments extended with an entry for each proof randomness, and leaves the other entries unchanged; the expression translation $\llbracket \_ \rrbracket_1$ takes H and a as free variables and returns their updated values of the form $extend \ldots (extend\ H\ E_1) \ldots E_n$, with one exponential expression E_i for each assertion in e, and $a, a_1, \ldots a_m$ for each additional evidence a_j produced by e.

We now explain the main cases of the first-pass translation. Public expressions are (still) unaffected. Note that they may includes public expressions generated by the shared translation, such as products of commitments. Affine private expressions are translated homomorphically, adding a corresponding linear expression on the proof randomness. Private exponential computations yields evidence that must be communicated to the verifier; we add their results to a. More complex private expression are supplemented with the sampling of a fresh message randomness for their result—we rely on the assertions introduced by the shared translation to prove those expressions.

Assertions of equations of the form $e_P = e_x$ are transformed into extensions of the global-challenge computation. The left-hand-side must be a public expression, and is discarded. The right-hand-side must be an expression on private variables. Let e_t be this expressions obtained by replacing every variable x with t_x. The translation computes it, and extends H with the result. Declassifications are similarly translated: the declassified value x is added to a, and the hash is extended with g^{t_x} to link it to its computed proof randomness (as if we were translating $\textbf{assert}\, g^x = g^x$). Continuing with our example, we give below the expressions e_1, obtained by translating the shared-translation of the discriminant query, after removing the unnecessary commitment C_d. (This code has been rearranged for simplicity; the intermediate code produced by applying the translation rules appears in Figure 8.)

```
$\textbf{let}\ p = z{*}z$
$\textbf{let}\ t_p = random()$
$\textbf{let}\ o' = oz * z$
$\textbf{let}\ t_{o'} = random()$
$\textbf{let}\ H = extend\ H\ ((C_z)^{t_z} *_G g^{-t_p} *_G h^{-t_{o'}})$
$\textbf{let}\ op = sample()$
$\textbf{let}\ t_{op} = random()$
```

```
$\mathbf{let}\ C_p = g^p *_G h^{op}$
$\mathbf{let}\ H = extend\ H\ (g^{t_p} *_G h^{t_{op}})$
$\mathbf{let}\ d = p - 4 * x * y$
$\mathbf{let}\ t_d = t_p - 4 * x * t_y$
$\mathbf{let}\ H = extend\ H\ g^{t_d}$
$\mathbf{let}\ a = a, (p,t_p), (o',t_{o'}), (op, t_{op}), C_p, d$
$H, a, d$
```

Response Translation In the second pass, after completing the computation of the global challenge c, we revisit the collected evidence a, and we replace every pair of a private value x and associated proof randomness t_x with the response $r_x = t_x - c * x$. This pass is defined by induction on the *type* of a, produced by the first-message translation, which indicates where those pairs are. (Technically, this pass also needs to re-balance nested tuples, as the prover produces $(\dots(a_0, a_1), a_2, \dots, a_n)$ whereas the verifier consumes $(a_0, (a_1, (\dots a_n)\dots))$; we omit those details.) Continuing with the discriminant prover, the resulting evidence $a : \delta$ binds the series of variables

$$(z, t_z), (oz, t_{oz}), (p,t_p), (o',t_{o'}), (op, t_{op}), C_p, d$$

and thus $\llbracket\delta\rrbracket_2$ simply computes the responses for the five pairs of secret and associated proof randomness:

```
$\llbracket\delta\rrbracket_2 \triangleq \mathbf{let}\ (z,t_z), (oz,t_{oz}), (p,t_p), (o',t_{o'}), (op, t_{op}), C_p, d = a$
        $\mathbf{let}\ r_z = t_z - c * z$
        $\mathbf{let}\ r_{oz} = t_{oz} - c * oz$
        $\mathbf{let}\ r_p = t_p - c * p$
        $\mathbf{let}\ r'_o = t_{o'} - c * o'$
        $\mathbf{let}\ r_{op} = t_{op} - c * op$
        $(r_z, r_{oz}, r_p, r_{o'}, r_{op}, C_p, d)$
```

*Top-Level Prover Translation (*P*)* We arrive at the following top-level prover, given here for the discriminant query. (See Figure 6 for the general case.) This prover relies on data sources extending both private source inputs y and z with an opening, a commitment, and a signature on that commitment

```
$x, (y, oy, C_y, \sigma_y), (z, oz, C_z, \sigma_z) \rightarrow$
    $\mathbf{let}\ H = H_0$
    $\mathbf{let}\ t_z = random()$
    $\mathbf{let}\ t_{oz} = random()$
    $\mathbf{let}\ a = (z,t_z), (oz,t_{oz})$
    $\mathbf{let}\ H = extend\ H\ g^{t_z} h^{t_{oz}}$
    $\mathbf{let}\ H, a\colon \delta, d = \llbracket \llbracket\theta\rrbracket \vdash \llbracket e\rrbracket \rrbracket_1$ // phase 1 detailed above
    $\mathbf{let}\ c = finalize\ H\ \mathbf{in}$
    $\mathbf{let}\ a = \llbracket\delta\rrbracket_2$                        // phase 2 detailed above
    $x, (C_y, \sigma_y), (C_z, \sigma_z), a, c$
```

In the top-level prover translation, $\llbracket\theta\rrbracket_{\mathsf{D}}$ is the tuple type of the (extended) provided data, and $\llbracket\theta\rrbracket_{\mathsf{pub}}$ is an expression that extracts their public parts (including the plain signatures, excluding lookup tables). The type δ of the additional evidence depends on the first-pass of the translation, and is used to drive the second part. In-between, the final value $H : hash$ is finalized into the global challenge $c : num$. Finally, the message passed from the prover to the verifier consists of (1) the public parts of the input data and of the result; (2) the additional evidence for proving this result; and (3) the global challenge for verifying this proof.

Verifier Translation Also following the shared translation, the prover translation leaves the public parts of the query unchanged, and it incrementally re-computes the challenge using the responses and additional evidence prepared by the prover for the private parts of the query. Figure 7 gives the compositional translation applied to the result of the shared translation, and the top-level translation $\llbracket \_ \rrbracket_{\mathsf{VERIFIER}}$. In combination, the verifier is defined as $\mathsf{V} \triangleq \llbracket \llbracket\theta \vdash e\rrbracket \rrbracket_{\mathsf{VERIFIER}}$.

Compositional translation $\llbracket \_ \rrbracket_{\mathsf{v}}$ In the verification pass, H is the public hash incrementally re-computing the global challenge, a is the received evidence consumed by the verifier, and every private variable x is replaced with a (public) response variable r_x—the type translation $\llbracket\rho\rrbracket_{\mathsf{v}}$ performs this replacement. In combination with the shared translation, every private source expression now yields a tuple of the form r_x, r_{ox}, C_x where r_x and r_{ox} are (presumably) responses associated with the exponents committed to C_x. (Again, all these values are actually optional in the compiler.)

The verifier expression $\llbracket v\rrbracket e$ takes free variables H and a, and additionally returns the updated H and the rest of a. Public expressions are unchanged. Private expressions are discarded, and replaced with response expressions, either computed (for affine expressions) or read off the evidence a (for more complex expressions). Note that the translation of affine expressions includes a term $-c * a_0$ for the constant, to ensure that, given correct responses for its free variables, the translation of an expression also produces a correct response.

Assertions of equations of the form $e_P = e_x$ are translated to hash computation, by computing the expression $(e_P)^c * e_r$, where e_r is obtained from e_x by replacing every variable x with r_x and extending H with the result. Declassifications $\downarrow x$ are similarly translated by reading x off the evidence a and extending the hash with g^{x+c*r_x}.

For instance, continuing with the discriminant, the (simplified) verifier translation $\llbracket \llbracket\rho\rrbracket \vdash \llbracket e_d\rrbracket \rrbracket_{\mathsf{v}}$ is

```
$\mathbf{let}\ r_p, r_{o'}, r_{op}, C_p, d, a = a$
$\mathbf{let}\ H = extend\ H\ ((C_z)^{t_z} *_G g^{-r_p} *_G h^{-r_{o'}})$
$\mathbf{let}\ H = extend\ H\ (g^{r_p} *_G h^{r_{op}})$
$\mathbf{let}\ r_d = r_p - 4 * x * r_y$
$\mathbf{let}\ H = extend\ H\ (C_p)^c *_G g^{r_d}$
$H, a\ ,d$
```

Top-Level Verifier We finally give the top-level translation, also for our sample discriminant query. (See Figure 7 for additional details.)

```
$x, C_y, \sigma_y, C_z, \sigma_z, a, c \rightarrow$
    $\mathit{verify}\ vk_y\ C_y\ \sigma_y$;
    $\mathit{verify}\ vk_z\ C_z\ \sigma_z$;
    **let** $r_z, r_{\sigma z}, a = a$
    **let** $H = \mathit{extend}\ H_0\ C_z^c *_G g_z^r *_G h^{r_{\sigma z}}$
    **let** $H, a, d = [\![ [\![\theta]\!] \vdash [\![e]\!] ]\!]_{\mathsf{V}}$    // translation detailed above
    $\mathit{check}\ c = \mathit{finalize}\ H$
    $d$
```

The prover first verifies the signatures on the two received commitments for y and z; it starts the challenge re-computation on the representation equation for input z (since we need a response for z an σz to check the proof of the square z^2), then proceeds with the verification for the query expression; it checks that the received and re-computed challenges match; it finally returns the public result d (unless of course *verify* or *check* raised an error.)

6 Security Theorems

Consider a well-typed ZQL source query $\mathsf{Q} \triangleq \theta \rightarrow\downarrow e$, with t input variables $\theta = (x_i : \tau_i)_{i=0..t-1}$, that declassifies only its result and its translation $(\mathsf{S}, (\mathsf{K}_i, \mathsf{D}_i)_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$. We give our main results based on the definitions of §3. We provide proof outlines in Appendix A, including a discussion of type-based verification of the compiled protocol. For functional correctness and soundness, we also suppose that there is no source-program overflow—formally, integers and their operations are computed modulo q.

Theorem 1 (Functional Correctness)
$(\mathsf{S}, (\mathsf{K}_i, \mathsf{D}_i)_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$ is correct.

Theorem 2 (Perfect Privacy)
$(\mathsf{S}, (\mathsf{K}_i, \mathsf{D}_i)_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$ is $(t, 0)$-private.

Our soundness theorem below is in the random-oracle model, requiring that *extend* and *finalize* are independent random oracles. It assumes that the Discrete Logarithm (DL) and Strong Diffie Hellman (SDH) assumptions hold—to guarantee the security of commitments and CL-signatures, respectively—and assuming that the ℓ_{CMA} conventional signatures primitives of data-sources are chosen message attack secure (CMA).

Theorem 3 (Computational Soundness)
$(\mathsf{S}, (\mathsf{K}_i, \mathsf{D}_i)_{i=0..\ell-1}, \mathsf{P}, \mathsf{V})$ is (t, ε)-sound, where the execution time t and success probability ε are respectively lower- and upper-bounded by the corresponding parameters of the assumptions.

Concretely, let t_{DL}, t_{SDH}, $t_{CMA}S$ and ε_{DL}, ε_{SDH}, ε_{CMA} be those parameters, for large enough bounds on the number of calls to their primitives. If $t < t_{CMA} - t_{red1}$, $t < (t_{DL} - t_{red2})/2$, and $t < (t_{SDH} - t_{red3})/2$, where the t_{redi} are small constants, then $\varepsilon < \ell_{CMA} \cdot \varepsilon_{CMA} + Q \cdot \sqrt{\varepsilon_{DL} + (\ell - \ell_{CMA}) \cdot \varepsilon_{SDH}} + Q^2/q$, where Q is the number of random oracle queries made by $\mathscr{A}$ and q is the order of G and thus also the size of the challenge.

In contrast with our privacy theorem, which is information-theoretic, our concrete-security soundness theorem is somewhat more cumbersome than the asymptotic security theorems often found in theoretical cryptography, but it remains closer to reality, in which cryptographic primitives come with concrete security bounds, and thus provides guidance for configuring these primitives to achieve adequate security.

7 ZQL applications

The expressivity of ZQL stems from the ease with which the primitive operators can be composed to build larger queries. We illustrate this by providing queries for applications in prior literature.

In the setting of smart metering, a meter issues signed private readings, and a household needs to compute their bill on the basis of a public tariff policy that maps each reading to a fee over time. A number of custom privacy protocols have been proposed to do this [60, 53]. One such billing policy takes a table of public times and private readings, as well as a lookup table from readings to prices to be summed:

```
**let** *smart_meter_bill*
    ($R$: (int pub $*$ int) table) // *time*, *reading*
    ($T$: (int $*$ int) lookuptable) = // *reading*, *fee*
    $\downarrow$ (**sum** ((*time*, *reading*) $\rightarrow$ **lookup** *reading* $T$) $R$)
```

The query looks up the non-linear price of each reading in the table T, using **lookup** and sums the results.

Another popular application in the literature involves pay-as-you-drive insurance schemes. Such schemes require drivers to fit a black box in their car that records their driving habits, and allow the insurer to compute a premium based on the safety of the driving, as well as distance or time. The use of zero-knowledge protocols to support such automotive settings, including road usage billing and tolling has been well established in the literature [7, 67, 49].

An example policy used by a UK auto insurance pilot scheme involves recording the segment of road traveled, the distance and the speed and use those to subtract "points" from a virtual driving license. Points are linked to the magnitude of speed violations on the road segments traveled. The insurance rate per mile is then computed as a function of the points subtracted, up to a threshold where the insurance becomes invalid. We can express such a policy in ZQL using a table for the recorded road segments used, and lookup tables to encode the speed limit of road segments, the penalty points per magnitude of violation, and finally the insurance premium for a certain number of points:

```
let pay_as_you_go
  (Segments : (int * int * int * int) table)
  (Limits : (int * int) lookuptable )
  (Penalties : (int * int) lookuptable )
  (Rates : (int * int) lookuptable ) =
  let points =
    sum ((time, road, speed, miles) →
            let limit = lookup road Limits
            lookup (speed − limit) Penalties) Segments
  let rate = lookup points Rates
  let miles =
    sum ((time, road, speed, miles) → miles) Segments
  ↓ (miles * rate)
```

The *pay as you go* application makes extensive use of lookup tables to simulate traditional database half-joins between tables. The values of these tables are largely arbitrary and related to the insurance policy.

The final example illustrates how ZQL lookups can be used to approximate functions on real numbers. A very common problem in privacy preserving protocols for location based services is to prove that the reading from a trusted sensor is at a certain distance from a specific location. For example privacy friendly theft prevention system may need to periodically prove that a trusted reading is within a certain distance from their (secret) home location [62]. Similar protocols can be of use for offender monitoring, curfew enforcement or tracking of trucks of goods. Previous work has proposed zero-knowledge distance protocols, such as [17].

The *gps distance* protocol takes as secret inputs the longitude and latitude of two points, as well as some precomputed tables, and returns an approximation of the distance between the two points in meters. The approximation used works for small distances under the assumption that the curvature of the earth is negligible. It still requires the computation of the trigonometric function $\cos(x/2)$. To achieve this, we assume the input longitude and latitudes are in the units $rad/10^5$, and that intermediate computations are precise to two decimal points.

```
let gps_distance (lat1: int) (lon1: int) (lat2: int) (lon2: int)
                 (hcos: (int * int) lookuptable )
                 (red: (int * int) lookuptable)
                 (dist: (int * int) lookuptable) =
  let latsum = lat1 + lat2
  // Table: $hcos(x) = \mathrm{round}(\cos((x)/2 \cdot 10^5) \cdot 10^2)$
  let hc = lookup latsum hcos
  let dlat = lat2 − lat1
  let dlon = lon2 − lon1
  let lon_cos = dlon * hc
  // Table: $red(x) = \mathrm{round}(x/10^2) in (\mathrm{rad}/10^5)^2$
  let r2 = lookup lon_cos red
  let squares = dlat*dlat + r2
  // Table: $dist(x) = \mathrm{round}(\sqrt{x} \cdot R/10^5)$
  // where R is earth's radius (meters).
  ↓ (lookup squares dist)
```

In this example, lookups are used to approximate real functions, including trigonometric functions and division which is not yet natively supported. The *hcos* table has a large domain (of about 1 million items) but can be reused across multiple operations. Other tables have a relatively small domain related to the distances of the points compared.

8 Discussion

Prototype implementation & limitations Our compiler uses the language development and testing facilities of F#: we program source queries as (a small subset of) F#, then extract the ZQL abstract syntax tree (AST) through reflection. The compilation pipeline performs ZQL type-checking, applies the shared translation, and finally produces the data-source, prover and verifier code. Each of these steps operates on well-typed ZQL expressions. This enables us to share many optimizations as ZQL-to-ZQL transformations.

Besides standard optimizations, the compiler supports a more general variant of **lookup** primitive, named **find**, that returns any lookup-table row that meets a condition expressed as a boolean expression on the whole content of the row. This provides more flexibility on the use of lookup tables, but its compilation is more complex.

In addition to cryptographic code, ZQL also synthesizes a custom marshaller and un-marshaller for the cryptographic evidence and results of the query. Following the ZQL approach, this code is specialized and compiled for a specific proof. Hence, the size and location of all fields, parametrized on the input table lengths, in known at compile time and there is no need to rely on a general-purpose parser, a component that is traditionally a source of security flaws.

We support three distinct compiler back-ends:

Concrete F# The main branch of the compiler transforms and compiles the final ZQL data source, prover and verifier into F# code, linked either to the standard .NET big integer libraries, or to proprietary managed libraries that support pairing based cryptography.

Symbolic F# The second branch of the compiler is linked against symbolic execution libraries for all the operators and primitives. Interestingly, since the F# branch makes extensive use of abstract types in the final prover and verifier, there is no need to write a separate symbolic execution environment: the mathematical functions can simply be replaced with equivalents computing on symbolic polynomials. The resulting code jointly computes the execution time and the proof size, as polynomial expressions of the input lengths and the unit costs of each cryptographic operation. We use symbolic execution to predict the performance of the compiler, and hope to use it in the future to chose between alternative optimization strategies at compile time.

Concrete C++ Finally, we support compilation of the verifier to native C++ code, linked with high performance native big integer libraries. This branch involves transforming the functional ZQL verifier and unmarshaller code into an imperative program and optimizing it using standard low-level techniques such as removing dead code, removing spurious copies, and minimizing memory re-allocations. The resulting native program takes a proof as an input, and outputs the verified result. The native branch does not support on-the-fly compilation and execution, and currently works for RSA groups only. Yet the resulting binary can be easily deployed where .NET runtimes are not available.

The process of compiling a query remains fast even on small devices. Thus, a service could simply send ZQL queries to the user, to be reviewed, compiled, then executed locally. To this end, our compiler also has an API that takes source ZQL ASTs, compiles them to F#, then also compiles and dynamically load the resulting F# code. This is likely to be faster, cheaper, safer and more reliable than providing custom binaries every time the query is updated.

The prototype compiler is still subject to limitations. For instance, some optimizations, such as moving declassifications up in the dataflow to minimize the size of the Σ-protocol, or batching some exponential computations, could be applied more aggressively.

Performance Evaluation Table 1 illustrates the performance of ZQL code for the three applications presented in Section 7. It provides the execution time for the prover and verifiers, as well as the size of the proof, for different security parameters of RSA (1024 bits, 2048 bits) and the pairing based signatures. The *smart_meter_bill* readings table is of size $\ell_{read} = 5$ and the *pay_as_you_go* query road segments table is of size $\ell_{seg} = 25$. This means that for the 1024 bit RSA branch, the prover can process a meter reading every $\sim 120mS$ or a segment of road every $\sim 360mS$. The proof size for the pairing based branch is $\sim$ 755 bytes per reading and $\sim$ 1921 bytes per segment. As expected, the pairing based proofs are more compact than their RSA counterparts for the same levels of security. Prover timings take into account the generation of random numbers. We note that these numbers, while slow by the standards of non-privacy friendly computation, are perfectly adequate for computing bills and insurance premiums in real time.

Besides the main F# backend we experimented with a C++ back-end that compiles to a native verifier. Although more performant in absolute terms, the native verifier is not significantly faster than its F# counterpart. The RSA 1024 bit computation of the *pay_as_you_go* verifier took $4,290mS$ as compared with the F# backend using native big integer binding that took $5,111mS$. Profiling the C++ execution indicates that more than 90% of the time is spent inside the modular multiplication function performing exponentiations. Thus, improving the performance of ZQL comes down to either faster exponentiations (through batching, multi-exponentiation or hardware), or reducing the number of operations required through more aggressive simplification of the cryptographic protocols.

Finally, table 1 illustrates the output of the symbolic execution engine on these three applications, in a configuration that measures the number of exponentiations (E), pairings ($\hat{e}$), and signature verification operations ($sigv$) in terms of the length of the input tables (ℓ_{read} and ℓ_{seg}), and ignore all other costs.

Where next? The current ZQL language is subject to some intrinsic limitations, and we are actively exploring options to overcome them.

Many of the limitations are cryptographic and could be overcome by applying more advanced protocols. For example, **lookup** and **find** are currently restricted to externally signed tables. Lookup tables based on accumulators [15] or vector commitments [31] would be more flexible and may reduce cost. At a lower level, table processing leads to many similar cryptographic operations in a data-parallel style. Batch proof and verification techniques and homomorphic signature schemes could speed them up [12]. Well known, zero-knowledge proofs for disjunctions, would allow branching statements. The shared translation could bundle multiple secrets per commitment. We note that chosing automatically the best encoding and technique, as well as compiling them in a compositional manner are challenging open problems.

On the language design side, we illustrated in §7 how functions can be approximated though lookups. ZQL could automate and optimize the process by compiling data sources that calculate and sign function-tables appropriately. Finally, by design, our source language shields programmers from cryptography, and this may hinder power-users that wish to customize our compilation scheme, or experiment with its variants. Similarly, some users may wish to rely on external, unverified procedures, and use ZQL only to validate their results. Advanced APIs exposing the internals of the ZQL compiler without breaking its invariants would help them.

References

[1] *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, 2010. USENIX Association.

[2] J. A. Akinyele, M. D. Green, and A. D. Rubin. Charm: A frame-

Examples (branch)	prover (mS)	verifier (mS)	proof size (Bytes)
smart meter bill (1024)	586	599	$6,106$
smart meter bill (2048)	$3,498$	$3,148$	$10,585$
smart meter bill (PB)	$1,374$	$2,092$	$3,773$
smart meter bill (symbolic)	$\mathsf{E}+16\cdot\mathsf{E}\cdot\ell_{read}+ 6\cdot\ell_{read}\cdot\hat{e}$	$6\cdot\mathsf{E}+14\cdot\mathsf{E}\cdot\ell_{read}+ 8\cdot\ell_{read}\cdot\hat{e}+sigv$	$67+\lvert h\rvert+\lvert sig\rvert+2\cdot\ell_{Ga}+ \ell_{Ga}\cdot\ell_{read}+22\cdot\ell_{read}+ 2\cdot\ell_{read}\cdot q+num+7\cdot q$
pay as you go (1024)	$5,314$	$5,111$	$57,368$
pay as you go (2048)	$32,442$	$30,859$	$100,099$
pay as you go (PB)	$8,305$	$12,261$	$28,819$
pay as you go (symbolic)	$15\cdot\mathsf{E}+40\cdot\mathsf{E}\cdot\ell_{seg}+ 12\cdot\ell_{seg}\cdot\hat{e}+6\cdot\hat{e}$	$29\cdot\mathsf{E}+35\cdot\mathsf{E}\cdot\ell_{seg}+ 16\cdot\ell_{seg}\cdot\hat{e}+8\cdot\hat{e}+sigv$	$167+\lvert h\rvert+\lvert sig\rvert+6\cdot\ell_{Ga}+ 4\cdot\ell_{Ga}\cdot\ell_{seg}+56\cdot\ell_{seg}+ 8\cdot\ell_{seg}\cdot q+num+23\cdot q$
gps dist (1024)	501	529	5044
gps dist (2048)	$3,017$	$2,889$	8629
gps dist (PB)	841	$1,253$	2751
gps dist (symbolic)	$60\cdot\mathsf{E}+18\cdot\hat{e}$	$71\cdot\mathsf{E}+24\cdot\hat{e}+4\cdot sigv$	$233+\lvert h\rvert+4\cdot\lvert sig\rvert+ 10\cdot\ell_{Ga}+33\cdot q$

Table 1: Performance for our three applications: runtime, and communicated proof sizes

work for rapidly prototyping cryptosystems. Cryptology ePrint Archive, Report 2011/617, 2011. `http://eprint.iacr.org/`.

[3] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 151–167. Springer, 2010. ISBN 978-3-642-15496-6.

[4] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 488–500. ACM, 2012. ISBN 978-1-4503-1651-4.

[5] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In R. D. Prisco and M. Yung, editors, *SCN 2006*, volume 4116 of *LNCS*, pages 111–125, Maiori, Italy, 2006. Springer.

[6] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. *19th Annual Network & Distributed System Security Symposium (NDSS12). Internet Society*, 2012.

[7] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Pretp: Privacy-preserving electronic toll pricing. In *USENIX Security Symposium* DBL [1], pages 63–78.

[8] E. Bangerter, J. Camenisch, and U. M. Maurer. Efficient proofs of knowledge of discrete logarithms and representations in groups with hidden order. In S. Vaudenay, editor, *Public Key Cryptography*, volume 3386 of *Lecture Notes in Computer Science*, pages 154–171. Springer, 2005. ISBN 3-540-24454-9.

[9] E. Bangerter, T. Briner, W. Henecka, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of sigma-protocols. In F. Martinelli and B. Preneel, editors, *EuroPKI*, volume 6391 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2009. ISBN 978-3-642-16440-8.

[10] E. Bangerter, S. Krenn, A.-R. Sadeghi, and T. Schneider. Yaczk: Yet another compiler for zero-knowledge. In *USENIX Security Symposium*, 2010.

[11] G. Barthe, C. F. B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella. Probabilistic relational verification for cryptographic implementations.

[12] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, pages 263–280, 2012.

[13] M. Bellare and O. Goldreich. On defining proofs of knowledge. In Brickell [22], pages 390–420. ISBN 3-540-57340-2.

[14] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[15] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT*, pages 274–285, 1993.

[16] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 17–32, 2008.

[17] T. S. Benjamin. Zero-knowledge protocols to prove distances. Personal communication, 2008.

[18] K. Bhargavan, C. Fournet, and A. D. Gordon. F7: refinement types for F#, Sept. 2008. Available from `http://research.microsoft.com/F7/`.

[19] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL 2010*. ACM Press, 2010.

[20] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In Franklin [46], pages 41–55. ISBN 3-540-22668-0.

[21] S. Brands. Rapid demonstration of linear relations connected by boolean operators. In *EUROCRYPT*, pages 318–333, 1997.

[22] E. F. Brickell, editor. *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, 1993. Springer. ISBN 3-540-57340-2.

[23] T. Briner. Compiler for zero-knowledge proof-of-knowledge protocols, 2004.

[24] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In S. Cimato, C. Galdi, and G. Persiano, editors, *SCN*, volume 2576 of *Lecture Notes in Computer Science*, pages 268–289. Springer, 2002. ISBN 3-540-00420-3.

[25] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Franklin [46], pages 56–72. ISBN 3-540-22668-0.

[26] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In B. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1296 of *LNCS*, pages 410–424. Springer Verlag, 1997.

[27] J. Camenisch and E. Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. Technical Report Research Report RZ 3419, IBM Research Division, May 2002.

[28] J. Camenisch, A. Kiayias, and M. Yung. On the portability of generalized schnorr proofs. In Joux [54], pages 425–442. ISBN 978-3-642-01000-2.

[29] J. Camenisch, M. Kohlweiss, and C. Soriente. Solving revocation with efficient update of anonymous credentials. In J. A. Garay and R. D. Prisco, editors, *SCN*, volume 6280 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2010. ISBN 978-3-642-15316-7.

[30] J. L. Camenisch. *Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem*. PhD thesis, ETH Zürich, 1998. Diss. ETH No. 12520, Hartung Gorre Verlag, Konstanz.

[31] D. Catalano and D. Fiore. Vector commitments and their applications. Cryptology ePrint Archive, Report 2011/495, 2011. `http://eprint.iacr.org/`.

[32] D. Chaum and T. P. Pedersen. Wallet databases with observers. In Brickell [22], pages 89–105. ISBN 3-540-57340-2.

[33] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[34] R. Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, University of Amsterdam, Amsterdam, 1997.

[35] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In *CRYPTO*, pages 424–441, 1998.

[36] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, pages 174–187, 1994.

[37] I. Damgård. On Σ-protocols, 2002. Available at `http://www.daimi.au.dk/~ivan/Sigma.ps`.

[38] I. Damgård and E. Fujisaki. An integer commitment scheme based on groups with hidden order. *IACR Cryptology ePrint Archive*, 2001:64, 2001.

[39] G. Danezis and B. Livshits. Towards ensuring client-side computational integrity. In C. Cachin and T. Ristenpart, editors, *CCSW*, pages 125–130. ACM, 2011. ISBN 978-1-4503-1004-8.

[40] Y. Dodis, T. Ristenpart, and T. Shrimpton. Salvaging merkle-damgård for practical applications. In Joux [54], pages 371–388. ISBN 978-3-642-01000-2.

[41] C. Dwork. Differential privacy: A survey of results. *Theory and Applications of Models of Computation*, pages 1–19, 2008.

[42] S. Faust, M. Kohlweiss, G. A. Marson, and D. Venturi. On the non-malleability of the fiat-shamir transform. In S. D. Galbraith and M. Nandi, editors, *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2012. ISBN 978-3-642-34930-0, 978-3-642-34931-7.

[43] U. Feige and A. Shamir. Witness indistinguishable and witness hiding protocols. In H. Ortiz, editor, *STOC*, pages 416–426. ACM, 1990. ISBN 0-89791-361-2.

[44] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In A. V. Aho, editor, *STOC*, pages 210–217. ACM, 1987. ISBN 0-89791-221-7.

[45] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[46] M. K. Franklin, editor. *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-22668-0.

[47] T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM, 1991.

[48] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In B. S. K. Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1997. ISBN 3-540-63384-7.

[49] F. D. Garcia, E. R. Verheul, and B. Jacobs. Cell-based roadpricing. In S. Petkova-Nikova, A. Pashalidis, and G. Pernul, editors, *EuroPKI*, volume 7163 of *Lecture Notes in Computer Science*, pages 106–122. Springer, 2011. ISBN 978-3-642-29803-5.

[50] I. Goldberg. Natural zero-knowledge embedding in c++. Personal communication, October 2011.

[51] O. Goldreich, S. Micali, and A. Wigderson. How to prove all np-statements in zero-knowledge, and a methodology of cryptographic protocol design. In A. M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1986.

[52] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1): 186–208, 1989.

[53] M. Jawurek, M. Johns, and F. Kerschbaum. Plug-in privacy for smart metering billing. In S. Fischer-Hübner and N. Hopper, editors, *PETS*, volume 6794 of *Lecture Notes in Computer Science*, pages 192–210. Springer, 2011. ISBN 978-3-642-22262-7.

[54] A. Joux, editor. *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-01000-2.

[55] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proceedings of USENIX Security 2004*, pages 287–302, 2004.

[56] U. M. Maurer. Unifying zero-knowledge proofs of knowledge. In B. Preneel, editor, *AFRICACRYPT*, volume 5580 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2009. ISBN 978-3-642-02383-5. URL `http://dblp.uni-trier.de/db/conf/africacrypt/africacrypt2009.html#Maurer09`.

[57] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security Symposium* DBL [1], pages 193–206.

[58] T. Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 1992. doi: 10.1007/3-540-48071-4_3.

[59] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '92*, volume 576 of *LNCS*, pages 129–140, 1992.

[60] A. Rial and G. Danezis. Privacy-preserving smart metering. In Y. Chen and J. Vaidya, editors, *WPES*, pages 49–60. ACM, 2011. ISBN 978-1-4503-1002-4.

[61] A. Rial and G. Danezis. Privacy-Preserving Smart Metering. In *Proceedings of the 11th ACM workshop on Privacy in the electronic society (WPES 2011)*, page 12, Chicago,IL,USA, 2011. ACM.

[62] T. Ristenpart, G. Maganis, A. Krishnamurthy, and T. Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with dhts. In *17th USENIX Security Symposium*, pages 275–290, 2008.

[63] C.-P. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[64] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 266–278. ACM, 2011. ISBN 978-1-4503-0865-6.

[65] M. Tompa and H. Woll. Random self-reducibility and zero knowledge interactive proofs of possession of information. In *FOCS*, pages 472–482. IEEE Computer Society, 1987.

[66] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In P. Ning and T. Yu, editors, *Proceedings of the 2007 ACM Workshop on Privacy in the Electronic Society, WPES 2007*, pages 99–107. ACM, 2007.

[67] C. Troncoso, G. Danezis, E. Kosta, J. Balasch, and B. Preneel. Pripayd: Privacy-friendly pay-as-you-drive insurance. *IEEE Trans. Dependable Sec. Comput.*, 8(5):742–755, 2011.

[68] H. Wee. Zero knowledge in the random oracle model, revisited. In *ASIACRYPT*, pages 417–434, 2009.

A Proof Outline of Security Theorems

PROOF OUTLINE FOR THEOREM 2 Consider the following sequence of game rewrites:

- *Game 0.* The same as the *Priv* game with $b = 0$.
- *Game 1.* The same as Game 0, except that it simulates the Σ-protocol by using the HVZK simulator and programming the random oracle.

  Game 0 and 1 are equivalent because of the zero-knowledge property.
- *Game 2.* The same as Game 1, except that it picks a random group element for A', generates trapdoor parameters for the Pedersen commitment, and computes commitments and openings by computing C first and then trapdoor opening them to the $b = 0$ values.

  Game 1 and 2 are equivalent because of the perfect blinding of A' and the trapdoor property of Pedersen commitments.
- *Game 3.* The same as Game 2, except that it trapdoor opens them to the $b = 1$ values.

  Game 2 and 3 are equivalent as R' is unaffected by any of the secret values.
- *Game 4.* The same as Game 3, except that it stops picking A' as random and generating trapdoor parameters for the Pedersen commitment and computes commitments honestly. The commitments and the A' values are not computed for $b = 1$ values.

  Game 3 and 4 are equivalent because of the perfect blinding of A' and the trapdoor property of Pedersen commitments.
- *Game 5.* The same as Game 4, except that it stops to simulates the Σ-protocol by using the HVZK simulator and programming the random oracle. The code of this game is the same as the *Priv* game with $b = 1$.

  Game 4 and 5 are equivalent because of the zero-knowledge property.

PROOF OUTLINE FOR THEOREM 3 We first check that the combined hash computation behaves like a random oracle, given that the individual functions *extend* and *finalize* behave like random oracles. This follows, e.g., from [40].

We then consider the following sequence of game rewrites:

- *Game 0.* The same as the *Unforgeability* game.
- *Game 1.* The same as Game 0, except that we abort if the proof contains a fresh signature for an uncorrupted signing key for non-lookup table inputs.

  We bound $|Pr[Game\ 1] - Pr[Game\ 0]|$, by ℓ_{CMA} times the probability of the best $t + t_{red1}$ algorithm breaking the unforgeability of the signature scheme.
- *Game 2.* The same as Game 1, except that we use a rewinding extractor and abort if extraction fails.

  $Pr[Game\ 1] \leq \sqrt{Pr[Game\ 2]} \cdot Q + Q^2/q$. This follows form the Forking Lemma. See [42, Theorem 3].
- *Game 3.* The same as Game 2, except that we abort if a linear combination of commitments, or a commitment in a multiplication proof is opened to a value different than it's expected value.

  This gives us a Pedersen commitment that is opened to two different values and we bound $|Pr[Game\ 3] - Pr[Game\ 2]|$, by the probability of the best $2t + t_{red2}$ algorithm for solving the discrete logarithm problem.
- *Game 4.* The same as Game 3, except that we abort if the proof contains a fresh signature for an uncorrupted signing key for lookup-table inputs. For this game $Pr[Game4] = 0$.

  We bound $|Pr[Game\ 4] - Pr[Game\ 3]|$, by $(\ell - \ell_{CMA})$-times the probability of the best $2t + t_{red3}$ algorithm breaking the unforgeability of the CL-signature scheme.

Security Types and Information Flows We briefly describe our use of more advanced types to structure our proofs of privacy and integrity. We refer to [19, 64, 11] for additional details.

Refinement types provide a convenient way to specify and automatically check the safety properties of intermediate computations in code. Each type τ can be refined by adding a formula φ that states a property of the values at that type. For example, $n : int\{0 \leq n < 100\}$ is the type of natural numbers smaller than 100. The formulas can mention any value in scope; during typechecking, every formula must be proved from the formulas available in the typing environment; this task is delegated to Z3, an automated theorem prover.

In our setting, for example, we *define* the type *x o commitment* as an abbreviation for the refinement type $C : elt_G\{C =_G g^x *_G h^o\}$ where G, g, and h in the formula are bound in our cryptographic libraries. Hence, for instance, to type the third component of $x+y, ox+oy, C_x * C_y$ as an $(x+y, ox+oy)$ *commitment*, after unfolding the type abbreviation for C_x, C_y, and $C_x * C_y$, our typechecker automatically generates and proves

$$C_x =_G g^x h^{ox} \,\&\&\, C_y =_G g^y h^{oy} \Longrightarrow C_x C_y =_G g^{x+y} h^{ox+oy}$$

Crucially, refinements are used only for static verification, and they are erased after typechecking. Hence, a commitment to x is just an element of G, and any occurence of x in refinement formulas does not affect its privacy. In particular, the verifier can still reason about the integrity of secrets using existentially-quantified variables in refinements.

Relational refinements types similarly provide a technique to specify and check equivalences between two runs of the same program on related (but possibly different) inputs. Each type τ can be relationally refined by adding a formula φ that relate the two values of the corresponding expression. For example, we *define* the type of public integers using the relational refinement type $x : int\{|x_0 = x_1|\}$, which states that the two values for x must coincide in both runs. Conversely, $x : int$, our type of private integers, does not state how these values x_0 and x_1 are related. With these definitions, it is clear that a public integer can be treated as a private integer (by just erasing the refinement), but the converse is not true, unless the typechecker can *prove* the equation $x_0 = x_1$ from assumptions in the environment. For example, addition in $\mathbb{F}_q$ is specified as *plus*: x: num $\rightarrow y$: num $\rightarrow z$: num { $z = x + y$ **mod** q } from which we can *derive* by logical subtyping, for instance, that z is public when both x and y are. More generally, this encoding of public values as relationally equal values also applies to cryptographic computations and tables, and provides a uniform basis for automatically verifying information-flow properties.

At the time of writing, we have relationally verified privacy and integrity for sample compiled programs using the F* tool, using type-based specification of their cryptographic primitives, but we still have to integrate this security verification step to the ZQL compiler.

Probabilistic sampling The main property of uniform random sampling, formalized using relational logic, is that it can be typed as $ox : num\{|ox_0 = F\ ox_1|\}$ for *any* one-to-one pure function F on its range—in our case integers modulo q.

In our compiled code, as we sample a fresh opening for a private variable x, we thus give it the type *x opening* defined as $ox : num\{|x_0 + \alpha ox_0 = x_1 + \alpha\ ox_1|\}$, where α is such that $h = g^\alpha$—since $\mathbb{F}_q$ is prime, this α exists, and can be used in specifications, even if it is not efficiently computable given g and h.

Similarly, as we sample a proof randomness for a private variable x, we give it the type *x rand* defined as $t_x : num\{|t_{x_0} - c_0 * x_0 = t_{x_1} - c_1 * x_1|\}$, where c is the global challenge—although this challenge c is not yet available as t_x is sample, at the specification level, we can rely on the random oracle to pre-sample its public random value ($c_0 = c_1$) returned as the proof hash is finalized.

Privacy by Typing We outline how to use relational typing to verify the prover code generated by our compiler and establish witness indistinguishability (Theorem 2). Intuitively, our main privacy property for the prover can be expressed as the relational typechecking judgment

$$[\![\theta]\!]_1\{|Q_0 = Q_1|\} \vdash \mathsf{P} : m : \tau\{|m_0 = m_1|\}$$

and its proof can be reduced to checking that each of the translation rules preserves relational typing. For instance, every source secret x included in m is declassified, so by hypothesis it has a relational type of that form; every commitment included in m is public (since it has type $C : elt_G\{C =_G g^x h^{ox}\}$ and the relational refinement formula $x_0 + \alpha ox_0 = x_1 + \alpha\ ox_1$ of ox logically implies $g^{x_0} h^{ox_0} = g^{x_1} h^{ox_1}$), and every response included in m is public (as can be shown from the relational refinement of the associated first-message randomness t_x).

B Prover and Verifier Translations Tables

First stage: $\llbracket x:\tau,\rho \rrbracket_1 = x:\tau, \llbracket \rho \rrbracket_1$ when x public (including all group elements)

$\llbracket x:\tau,\rho \rrbracket_1 = x:\tau, t_x : x\ \mathit{witness}, \llbracket \rho \rrbracket_1$ when x private int or num

$\llbracket \Gamma \vdash e \rrbracket_1 = H, a, e$

when e public expression, that is, whose variables are all public in Γ.

$\llbracket \Gamma \vdash e \rrbracket_1 = \mathbf{let}\ C = e\ \mathbf{in}\ \mathit{extend}\ H\ C, (a, C), C$ when $\Gamma \vdash e : \mathit{elt}_G$ and e is not public

$\llbracket \Gamma \vdash a_0 + \sum_{i=1}^{n} a_i * x_i \rrbracket_1 = H, a, a_0 + \sum_{i=1}^{n} a_i * x_i, \sum_{i=1}^{n} a_i * t_{x_i}$

when the x_i are private and the a_i public:

$(\Gamma(a_i) = \mathit{pub\ num})_{i=0..n}, (\Gamma(x_i) = \mathit{num})_{i=1..n}$

$\llbracket \Gamma \vdash e \rrbracket_1 = \mathbf{let}\ a, \rho = e\ \mathbf{in}\ (\mathbf{let}\ t_{x_i} = \mathit{random}()\ \mathbf{in}\ )_{x_i} H, (a, \llbracket \rho \rrbracket_1), \llbracket \rho \rrbracket_1$

when $\Gamma \vdash e : \rho$ non-linear private expression (including assoc, random, opening...)

and x_i ranges over the private variables bound in ρ

$\llbracket \Gamma \vdash \mathbf{assert}\ e_C =_G e_x \rrbracket_1 = \mathit{extend}\ H\ e_t, a, \varepsilon$ when e_C public and e_x algebraic on private exponents

$\llbracket \Gamma \vdash \downarrow x \rrbracket_1 = \mathbf{let}\ a = a, x\ \mathbf{in}\ \mathit{extend}\ H\ g^{t_x}, a, x$

$\llbracket \Gamma \vdash \mathbf{let}\ \rho = e\ \mathbf{in}\ e_0 \rrbracket_1 = \mathbf{let}\ H, a, \llbracket \rho \rrbracket_1 = \llbracket \Gamma \vdash e \rrbracket_1\ \mathbf{in}\ \llbracket \Gamma, \rho \vdash e_0 \rrbracket_1$

$\llbracket \Gamma \vdash \mathbf{map}\,(\rho \to e)\, T \rrbracket_1 = \mathbf{let}\ H, T_a, T' = \mathbf{mapP}\,(H, \llbracket \rho \rrbracket_1 \to \mathbf{let}\ a = H_0\ \mathbf{in}\ \llbracket \Gamma, \rho \vdash e \rrbracket_1)\, H\, \llbracket \Gamma \vdash T \rrbracket_1\ \mathbf{in}$

$H, (a, T_a), T'$

$\llbracket \Gamma \vdash \mathbf{fold}\,(r:\tau, \rho \to e)\, r\, T \rrbracket_1 = \mathbf{let}\ H, T_a, r = \mathbf{foldP}\,(H, \llbracket r:\tau, \rho \rrbracket_1 \to \mathbf{let}\ a = H_0\ \mathbf{in}\ \llbracket \Gamma, r:\tau, \rho \vdash e \rrbracket_1)\, H\, \llbracket \Gamma \vdash r \rrbracket_1\, \llbracket \Gamma \vdash T \rrbracket_1\ \mathbf{in}$

$H, (a, T_a), r$

Second stage: $\llbracket \delta, \rho \rrbracket_2 = \mathbf{let}\ a, \llbracket \rho \rrbracket_1 = a\ \mathbf{in}\ \llbracket \delta \rrbracket_2, [\mathbf{let}\ r_x = t_x - c * x\ \mathbf{in}\ ]_x \llbracket \rho \rrbracket_{\mathsf{v}}$

where x ranges over the private variables bound in ρ

$\llbracket \delta, \delta'\ \mathit{table} \rrbracket_2 = \mathbf{let}\ a, A = a\ \mathbf{in}\ \llbracket \delta \rrbracket_2, \mathbf{map}\,(\delta' \to \llbracket \delta' \rrbracket_2)\, A$

$\llbracket \varepsilon \rrbracket_2 = \varepsilon$

$\llbracket \theta \to e \rrbracket_{\mathsf{PROVER}} = \llbracket \theta \rrbracket_{\mathsf{D}} \to$

$\mathbf{let}\ H = H_0\ \mathbf{in}\ \mathbf{let}\ a = ()\ \mathbf{in}$

// prove commitments for all private inputs (omitted)

$\mathbf{let}\ H : \mathit{hash}, a : \delta, r = \llbracket \theta \vdash e \rrbracket_1\ \mathbf{in}$

$\mathbf{let}\ c = \mathit{finalize}\ H\ \mathbf{in}$

$\llbracket \theta, r \rrbracket_{\mathsf{pub}}, \llbracket \delta \rrbracket_2, c$

Figure 6: Prover Translation

$$
\begin{aligned}
\llbracket x:\tau,\rho\rrbracket_{\mathsf{v}} &= x:\tau,\llbracket\rho\rrbracket_{\mathsf{v}} \text{ when } x \text{ public (including all group elements)}\\
\llbracket x:\tau,\rho\rrbracket_{\mathsf{v}} &= r_x:(c,x)\ \mathit{response},\llbracket\rho\rrbracket_{\mathsf{v}} \text{ when } x \text{ private}\\[1ex]
\llbracket \Gamma\vdash e\rrbracket_{\mathsf{v}} &= H,a,e\\
&\quad \text{when } e \text{ public expression, that is, whose variables are all public in } \Gamma.\\
\llbracket \Gamma\vdash e\rrbracket_{\mathsf{v}} &= \textbf{let } C,a=a \textbf{ in } \mathit{extend}\ H\ C,a,C \qquad \text{when } \Gamma\vdash e:\mathit{elt}_G \text{ and } e \text{ is not public}\\
\llbracket \Gamma\vdash a_0+\textstyle\sum_{i=1}^{n} a_i*x_i\rrbracket_{\mathsf{v}} &= H,a,-c*a_0+\textstyle\sum_{i=1}^{n} a_i*r_{x_i}\\
&\quad \text{when the } x_i \text{ are private and the } a_i \text{ public:}\\
&\quad (\Gamma(a_i)=\mathit{pub\ num})_{i=0..n},(\Gamma(x_i)=\mathit{num})_{i=1..n}\\
\llbracket \Gamma\vdash e\rrbracket_{\mathsf{v}} &= \textbf{let } a,\llbracket\rho\rrbracket_{\mathsf{v}}=a \textbf{ in } H,a,\llbracket\rho\rrbracket_{\mathsf{v}}\\
&\quad \text{when } \Gamma\vdash e:\rho \text{ non-linear private-exponent expression (including assoc, random, opening...)}\\
&\quad \text{and } \rho \text{ binds private exponents and public elements}\\
\llbracket \Gamma\vdash \mathit{assert}\ e_C =_G e_x\rrbracket_{\mathsf{v}} &= \mathit{extend}\ H\ ((e_C)^c *_G \llbracket e_x\rrbracket_{\mathsf{v}}),a,\varepsilon \qquad \text{when } e_x \text{ algebraic on private exponents}\\
\llbracket \Gamma\vdash\downarrow x\rrbracket_{\mathsf{v}} &= \textbf{let } x,a=a \textbf{ in}\\
&\quad \mathit{extend}\ H\ g^{c*x+r_x},a,x\\[1ex]
\llbracket \Gamma\vdash \textbf{let } \rho=e \textbf{ in } e_0\rrbracket_{\mathsf{v}} &= \textbf{let } H,a,\llbracket\rho\rrbracket_{\mathsf{v}}=\Gamma\vdash\llbracket e\rrbracket_1 \textbf{ in } \llbracket\Gamma,\rho\vdash e_0\rrbracket_{\mathsf{v}}\\
\llbracket \Gamma\vdash \textbf{map}\,(\rho\to e)\,T\rrbracket_{\mathsf{v}} &= \textbf{let } A,a=a \textbf{ in}\\
&\quad \textbf{let } H,R=\textbf{mapV}\,(H,a,\llbracket\rho\rrbracket_{\mathsf{v}}\to\llbracket\Gamma,\rho\vdash e\rrbracket_{\mathsf{v}})\,H\,A\,\llbracket\Gamma\vdash T\rrbracket_{\mathsf{v}} \textbf{ in}\\
&\quad H,a,R\\
\llbracket \Gamma\vdash \textbf{fold}\,(r:\tau,\rho\to e)\,r\,T\rrbracket_{\mathsf{v}} &= \textbf{let } A,a=a \textbf{ in}\\
&\quad \textbf{let } H,r=\textbf{foldV}\,(H,a,\llbracket r:\tau,\rho\rrbracket_{\mathsf{v}}\to\llbracket\Gamma,r:\tau,\rho\vdash e\rrbracket_{\mathsf{v}})\,H\,a\,\llbracket\Gamma\vdash r\rrbracket_{\mathsf{v}}\ \llbracket\Gamma\vdash T\rrbracket_{\mathsf{v}} \textbf{ in}\\
&\quad H,a,r\\[1ex]
\llbracket\theta\to e\rrbracket_{\mathsf{VERIFIER}} &= \llbracket\theta,r\rrbracket_{\mathsf{pub'}},a,c\to\\
&\qquad \text{// check plain signatures and commitment proofs for all private inputs (omitted)}\\
&\qquad \textbf{let } H=H_0 \textbf{ in}\\
&\qquad \textbf{let } H,a,r=\llbracket\Gamma\vdash e\rrbracket_{\mathsf{v}} \textbf{ in}\\
&\qquad \mathit{check}\ c=\mathit{finalize}\ H;\\
&\qquad r
\end{aligned}
$$

Figure 7: Verifier Translation

We translate

```
let $p, op, C_p =$
   let $p = z * z$ in
   let $o' = oz * z$ in
   assert $1 = (C_z)^z *_G g^{-p} *_G h^{-o'}$;                 $(E_1)$
   let $op = sample()$ in
   let $C_p = g^p *_G h^{op}$ in
   assert $C_p = g^p *_G h^{op}$;                               $(E_2)$
   $(p, op, C_p)$ in
let $d =$
   $p - 4 * x * y$ in
$\downarrow d$
```

to

```
let $H$, $a$, $p$, $t_p$, $op$, $t_{op}$, $C_p$ =
  let $H$, $a$, $p$, $t_p$ =
    let $p = z*z$
    let $t_p$ = random()
    $H$, ($a$,($p$,$t_p$)), $p$, $t_p$ in
  let $H$, $a$, $o'$, $t_{o'}$ =
    let $o' = oz * z$
    let $t_{o'}$ = random()
    $H$, ($a$,($o'$,$t_{o'}$)), $o'$, $t_{o'}$
  let $H$, $a$ = extend $H$ ($(C_z)^{t_z} *_G g^{-t_p} *_G h^{-t_{o'}}$), $a$
  let $H$, $a$, $op$, $t_{op}$ =
    let $op$ = sample()
    let $t_{op}$ = random()
    $H$, ($a$,($op$, $t_{op}$)), $op$, $t_{op}$
  let $H$, $a$, $C_p$ =
    let $C_p = g^p *_G h^{op}$
    $H$, ($a$, $C_p$), $C_p$
  let $H$, $a$ = extend $H$ ($g^{t_p} *_G h^{t_{op}}$), $a$
  $H$, $a$, ($p$,$t_p$), ($op$,$t_{op}$), $C_p$
let $H$, $a$, $d$, $t_d$ =
  let $d = p - 4 * x * y$
  let $t_d = t_p - 4 * x * t_y$
let $a$ = $a$, $d$
let $H$, $a$ = extend $H$ $g^{t_d}$, $a$
$H$, $a$, $d$
```

Figure 8: Sample prover translation for the discriminant.

```
let rec mapP
   (f: hash → 'x → hash ∗ α ∗ 'y)
   (\H:hash) (X:'x list) =
   match X with
   | x::X → let H,a,y = f H x
            let H,A,Y = mapP f H X
            (H, a::A, y::Y)
   | [] → H,[],[]
let rec foldP
   (f: hash → 'r → 'x → hash ∗ α ∗ 'r)
   (H:hash) (r:'r) (X:'x list) =
   match X with
   | x::X → let H,a,r' = f H r x
            let H,A,r'' = foldP f H r' X
            (H, a::A,r'')
   | [] → H,[],r
```

Figure 9: Auxiliary F# iterators in the Prover translation

```
let rec mapV
   (f: hash → α → 'x → hash ∗ unit ∗ 'y)
   (\H:hash) (A:α list) (X:'x list) : hash ∗ 'y list =
   match A, X with
   | (a::A),(x::X) → let \H,(),y = f \H a x
                     let \H,Y = mapV f \H A X
                     (\H, y::Y)
   | [],[] → \H,[]
   | _ → failwith "table length mismatch"
let rec foldV
   (f: hash → α → 'r → 'x → hash ∗ unit ∗ 'r)
   (\H:hash) (A:α list) (r:'r) (X:'x list) : hash ∗ 'r =
   match A,X with
   | (a::A),(x::X) → let \H,(),r' = f h a r x
                     foldV f \H A r' X
   | [],[] → \H,r
   | _ → failwith "table length mismatch"
```

Figure 10: Auxiliary F# iterators in the Verifier translation